MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

May 1974

A. I. Memo 305 Logo Memo 10

SUMMARY OF MYCROFT: A SYSTEM FOR UNDERSTANDING SIMPLE PICTURE PROGRAMS*

Ira P. Goldstein

ABSTRACT

A collection of powerful ideas--description, plans, linearity, insertions, global knowledge and imperative semantics--are explored which are fundamental to debugging skill. To make these concepts precise, a computer monitor called MYCROFT is described that can debug elementary programs for drawing pictures. The programs are those written for LOGO turtles.

*The first section of this paper will appear in the Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior to be held at the University of Sussex, July 1974.

This work was supported in part by the National Science Foundation under grant number GJ-1049 and conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0005.

Table of Contents

DEBUGGING SIMPLE PICTURE PROGRAMS

1. INTRODUCTION

This paper reports on progress in the development of a monitor for debugging elementary programs. Such research is important both for its practical applications as well as for its investigation of concepts which are fundamental to programming skill. A computer monitor called MYCROFT has been designed that can repair simple programs for drawing pictures [Goldstein 1974]. The reasons to develop such monitors are:

1. to provide a more precise understanding of the nature of programming skills;

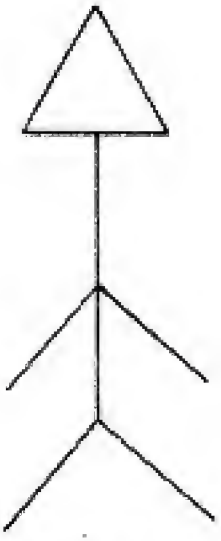
2. to facilitate the development of machines capable of debugging and expanding upon the programs given them by humans; and

3. to produce insight into the problem solving process so that it can be described more constructively to students.

MYCROFT is intended to supply occasional advice to a student to aid in the debugging of programs that go awry. (Just as the system's namesake, Mycroft Holmes, occasionally supplied advice to his younger brother Sherlock on particularly difficult cases.) In this interaction, the user supplies statements that describe aspects of the intended picture and plan, and the system fills in details of this commentary, diagnoses bugs and suggests corrections. In this paper, however, I shall not emphasize this interactive role. Instead, my primary purpose will be to describe MYCROFT as a model of the debugging process. This is reasonable since MYCROFT's utility as an advisor stems directly from its understanding of debugging skill.

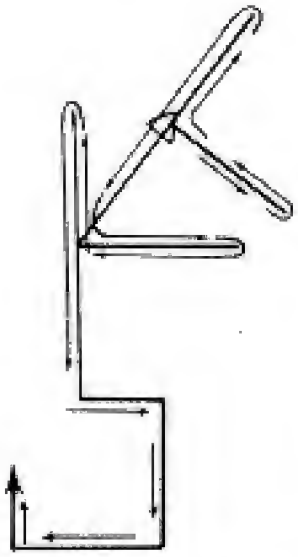
MYCROFT is able to correct the programs responsible for the bugged pictures shown in figures 1.1, 1.3, 1.4 and 1.5 so that the intended pictures are achieved. In this paper, the debugging of figure 1.1, a

typical example, will be thoroughly explained. Figures 1.3, 1.4 and 1.5 are corrected in analogous ways: see [Goldstein 1974] for details.



Intended MAN

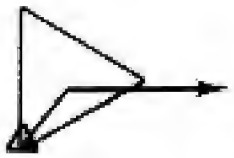
FIGURE 1.1



Picture drawn by NAPOLEON

FIGURE 1.2

INTENDED TREE



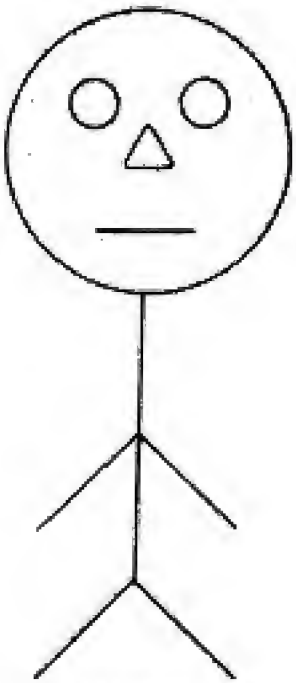
Picture drawn by
bugged TREE program

FIGURE 1.3

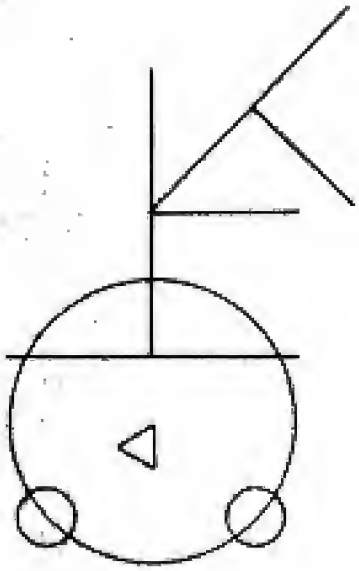
Intended WISHINGWELL

Picture drawn by bugged WISHINGWELL program

FIGURE 1.4



Intended FACEMAN



Picture drawn by bugged FACEMAN program

FIGURE 1.5

These pictures are drawn by program manipulation of a graphics device called the turtle which has a pen that can leave a track along the turtle's path. Turtles play an important role in the LOGO environment where children learn problem solving and mathematics by programming display turtles, physical turtles with various sensors, and music boxes [Papert 1971, 1972]. Turtle programs have proven to be an excellent starting point for teaching programming to children of all ages, and therefore provide a reasonable initial problem domain for building a program understanding system.

The context of MYCROFT's activity is the interaction of three kinds of description: graphical (i.e. the picture actually drawn), procedural (the turtle program used to generate the picture) and predicative (the collection of statements used to describe the desired scene). For MYCROFT, debugging is making the procedural description produce a graphical result that satisfies the set of predicates describing intent. Thus, debugging here is a process that mediates between different representations of the same object.

1.1 FLOWCHART OF THE SYSTEM

The organization of the monitor system is illustrated in figure 1.6. Input to MYCROFT consists of the user's programs and a model of the intended outcome. For the graphics world, the model is a conjunction of geometric predicates describing important properties of the intended picture. MYCROFT then analyzes the program, building both a Cartesian annotation of the picture that is actually drawn and a plan explaining the relationship between the program and model. (Any or all of the plan can be supplied directly by the user, thereby simplifying MYCROFT's task.)

FLOWCHART OF MYCROFT

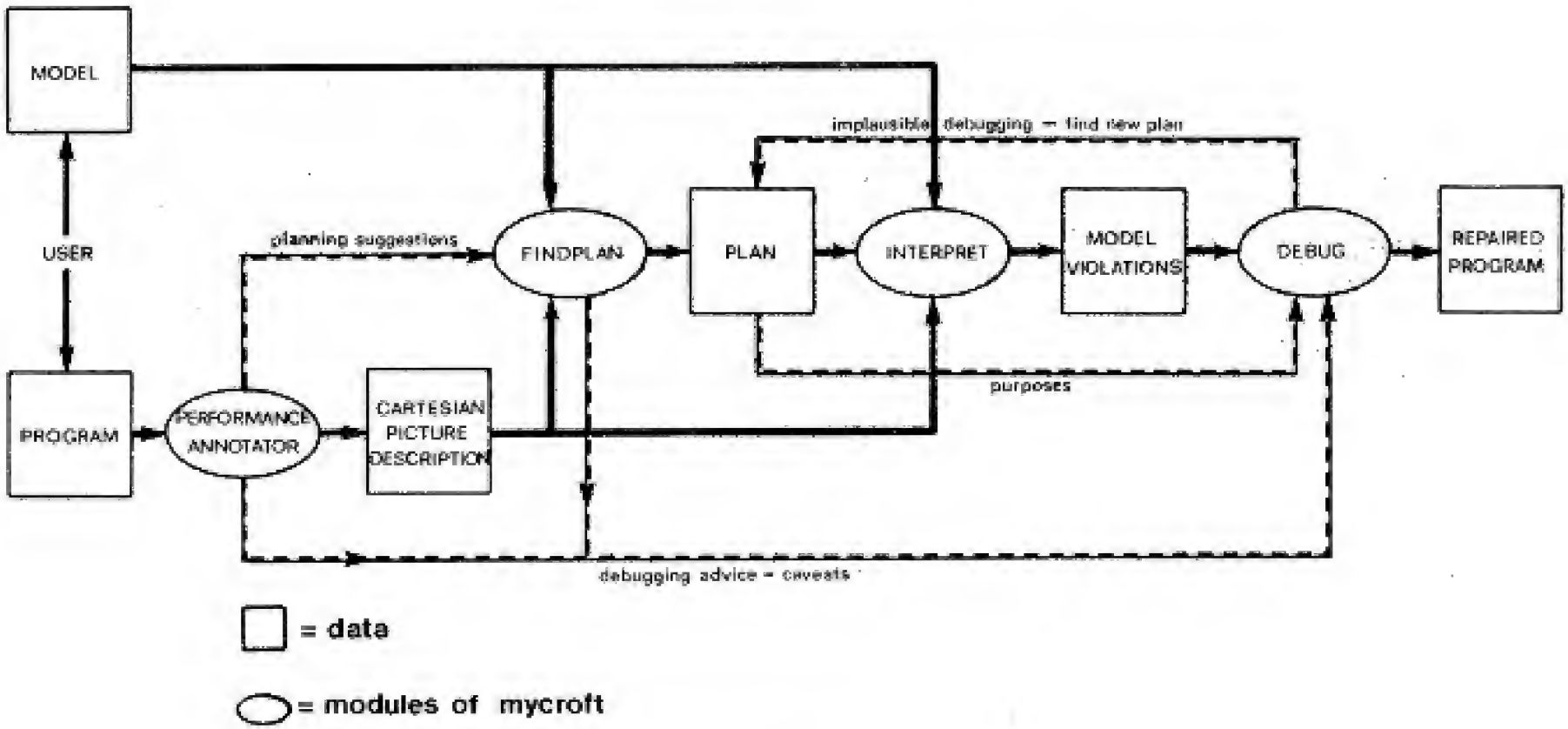


FIGURE 1.6

The next step is for the system to interpret the program's performance in terms of the model and produce a description of the discrepancies. These discrepancies are expressed as a list of the violated model statements. The task is then for the debugger to repair each violation. The final output is an edited turtle program (with copious commentary) which satisfies the model. (Occasionally, the plan that MYCROFT hypothesizes requires implausible repairs--for example, major deletions of user code--resulting in the debugger asking the plan-finder for a new plan.)

The remainder of this first section describes the debugging of NAPOLEON (figure 1.1) and introduces some important ideas about the nature of plans. Section 2 describes the annotator used to document the performance of turtle programs. Section 3 introduces the plan-finder and section 4 discusses the debugger. Section 5 concludes with suggestions for future research.

1.2 PICTURE MODELS

To judge the success of a program, MYCROFT requires as input from the user a description of intent. A declarative language has been designed to define picture models. These models specify important properties of the desired final outcome without indicating the details of the drawing process. The primitives of the model language are geometric predicates for such properties as connectivity, relative position, length and location. The following models are typical of those that the user might provide to describe figure 1.2.

```
MODEL MAN
M1 PARTS HEAD BODY ARMS LEGS
M2 EQUITRI HEAD
M3 LINE BODY
M4 V ARMS, V LEGS
M5 CONNECTED HEAD BODY, CONNECTED BODY ARMS, CONNECTED BODY LEGS
M6 BELOW LEGS ARMS, BELOW ARMS HEAD
END

MODEL V
M1 PARTS L1 L2
M2 LINE L1, LINE L2
M3 CONNECTED L1 L2 (VIA ENDPOINTS)
END

MODEL EQUITRI
M1 PARTS (SIDE 3) (ROTATION 3)
M2 FOR-EACH SIDE (= (LENGTH SIDE) 100)
M3 FOR-EACH ROTATION (= (DEGREES ROTATION) 120)
M4 RING CONNECTED SIDE
END
```

The MAN and V models are underdetermined: they do not describe, for example, the actual size of the pictures. The user has latitude in his description of intent because MYCROFT is designed only to debug programs that are almost correct. Therefore, not only the model, but also the picture drawn by the program and the definition of the procedure provide clues to the purpose of the program.

1.3 THE NAPOLEON EXAMPLE

MYCROFT is designed to repair a simple class of procedures called Fixed-Instruction Programs. These are procedures in which the primitives are restricted to constant inputs. Sub-procedures are allowed; however, no conditionals, variables, recursions or iterations are permitted. Given below are the three programs which drew figure 1.1--NAPOLEON, VEE, and TRICORN. The "<-" commentary is called the plan and was generated by MYCROFT to link the picture models--MAN, V and EQUITRI--to the programs.

```
TO NAPOLEON              <- (accomplish man)
10 VEE                   <- (accomplish legs)
20 FORWARD 100           <- (accomplish (piece 1 body))
30 VEE                   <- (insert arms body)
40 FORWARD 100           <- (accomplish (piece 2 body))
50 LEFT 90               <- (setup heading (for head))
60 TRICORN               <- (accomplish head)
END

TO VEE                   <- (accomplish v)
10 RIGHT 45              <- (setup heading for l1)
20 BACK 100              <- (accomplish l1)
30 FORWARD 100           <- (retrace l1)
40 LEFT 90               <- (setup heading for l2)
50 BACK 100              <- (accomplish l2)
60 FORWARD 100           <- (retrace l2)
END

TO TRICORN               <- (accomplish equitri)
10 FORWARD 50            <- (accomplish (piece 1 (side 1)))
20 RIGHT 90              <- (accomplish (rotation 1))
30 FORWARD 100           <- (accomplish (side 2))
40 RIGHT 90              <- (accomplish (rotation 2))
50 FORWARD 100           <- (accomplish (side 3))
60 RIGHT 90              <- (accomplish (rotation 3))
70 FORWARD 50            <- (accomplish (piece 2 (side 1)))
END
```

The turtle command FORWARD moves the turtle in the direction that it is currently pointed: RIGHT rotates the turtle clockwise around its axis. A complete description of LOGO can be found in [Abelson 1974], but is not needed here.

A Cartesian representation of the picture is generated by the annotator that describes the performance of the turtle program. The plan is used to bind sub-pictures to model parts. This allows MYCROFT to interpret the program with repect to the model and produce a list of violated model statements. MYCROFT produces the following list of discrepancies for NAPOLEON:

```
(NOT (LINE BODY))          ;The body is not a line.
(NOT (BELOW LEGS ARMS))    ;The legs are not below the arms.
(NOT (BELOW ARMS HEAD))    ;The arms are not below the head.
(NOT (EQUITRI TRICORN))    ;The head is not an equilateral triangle.
```

MYCROFT is able to correct these bugs and achieve the intended picture

using both planning and debugging knowledge.

1.4 PLANS

This section introduces a vocabulary for talking about the structure of a procedure which is useful for understanding both the design and debugging of programs. A main-step is defined as the code required to achieve a particular sub-goal (sub-picture). A preparatory-step consists of code needed to setup, cleanup or interface between main-steps. Thus, from this point of view, a program is understood as a sequence of main-steps and preparatory-steps. A similar point of view is found in [Sussman 1973]. The plan consists of the purposes linking main- and preparatory-steps to the model: in the turtle world, the purpose of main-steps is to accomplish (draw) parts of the model; and the purpose of preparatory-steps is to properly setup or cleanup the turtle state between main-steps or, perhaps, to retrace over some previous vector.

A modular main-step is a sequence of contiguous code intended to accomplish a particular goal. This is as opposed to an interrupted main-step whose code is scattered in pieces throughout the program. In NAPOLEON, the main-steps for the legs, arms and head are modular; however, the code for the body is interrupted by the insertion of the code for the arms into its midst. The utility of making this distinction is that modular main-steps can often be debugged in private (i.e. by being run independently of the remainder of the procedure) while interrupted main-steps commonly fail because of unforseen interactions with the interleaved code associated with other steps of the plan.

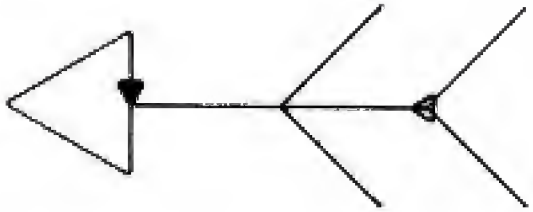
has two stages. The first is to break the task into independent sub-goals and design solutions (main-steps) for each. The second is then to combine these main-steps into a single procedure by concatenating them into some sequence, adding (where necessary) preparatory-steps to provide proper interfacing. The virtue of this approach is that it divides the problem into manageable sub-problems. A disadvantage is that occasionally there may be constraints on the design of some main-step which are not recognized when that step is designed independently of the remainder of the problem. Another disadvantage is that linear design can fail to recognize opportunities for sub-routinizing a segment of code useful for accomplishing more than one main-step. A linear plan will be defined as a plan consisting only of modular main-steps and preparatory steps: a non-linear plan may include interrupted main-steps.

1.5 LINEAR DEBUGGING

Linearity is a powerful concept for debugging as well as for designing programs. MYCROFT pursues the following linear approach to correcting turtle programs: the debugger's first goal is to fix each main-step independently so that the code satisfies all intended properties of the model part being accomplished. Following this, the main-steps are treated as inviolate and relations between model parts are fixed by debugging preparatory-steps. This is not the only debugging technique available to the system, but it is a valuable one because it embodies important heuristics (1) concerning the order in which violations should be repaired and (2) for selecting the repair-point (location in the program) at which the edit for each violation should be attempted.

Following this linear approach, MYCROFT repairs the crooked body and

the open head of NAPOLEON before correcting the BELOW relations. Repairing these parts is done on the basis of knowledge described in the next two sections. Let us assume for the remainder of this section that these property repairs have been made -- NAPOLEON appears as in figure 1.7 -- and concentrate on the debugging of the violated relations.



NAPOLEON with parts corrected

FIGURE 1.7

NAPOLEON with statement 15 as RIGHT 135

FIGURE 1.8

Treating main-steps as inviolate and fixing relations by modifying setup steps limits the repair of (BELOW LEGS ARMS) to three possible repair-points: (1) before the legs as statement 5, (2) before the first piece of the body as statement 15 and (3) before accomplishing the arms as statement 25. MYCROFT understands enough about causality to know that there is no point in considering edits following the execution of statement 30 to affect the arms or legs. The exact changes to be made are determined by imperative semantics for the model primitives. This is procedural knowledge that generates, for a given predicate and location in the program, some possible edits that would make true the

violated predicate. MYCROFT generally considers alternative strategies for correcting a given violation: it prefers those edits which produce the most beneficial side effects, make minimal changes to the user's code or most closely satisfy the abstract form of the plan.

For BELOW, the imperative semantics direct DEBUG to place the legs below the arms by adding rotations at the setup steps. More drastic modifications to the user's code are possible such as the addition of position setups which alter the topology of the picture; however, MYCROFT tries to be gentle to the turtle program (using the heuristic that the user's code is probably almost correct) and considers larger changes to the program only if the simpler edits do not succeed. The first setup location considered is the one immediately prior to accomplishing the arms. Inserting a rotation as statement 25, however, does not correct the violation and is therefore rejected. The next possible edit point is as statement 15. Here, the addition of RIGHT 135 makes the legs PARTLY-BELOW the arms and produces figure 1.8. This edit is possible but is not preferred both because the legs and arms now overlap and because the legs are not COMPLETELY-BELOW the arms. MYCROFT is cautious, being primarily a repairman rather than a designer, and is reluctant to introduce new connections not described in the model. Also, given a choice, MYCROFT prefers the most constrained meaning of the model predicate. If the user had intended figure 1.8, then one would expect the model description to include additional declarations such as (CONNECTED LEGS ARMS) and (PARTLY-BELOW LEGS ARMS).

Adding RIGHT 90 as statement 5 achieves (COMPLETELY-BELOW LEGS ARMS) and the NAPOLEON program now produces the intended picture (figure 1.2). This correction has beneficial side effects in also establishing the proper relationship between the head and arms, confirming for MYCROFT

that the edit is reasonable, since a particular underlying cause is often responsible for many bugs. Thus the result of (DEBUG (BELOW LEGS ARMS)) is:

```
5 RIGHT 90 <- (setup heading such-that (below legs arms)
                                       (below arms head))
              (assume (= (entry heading) 270))
```

The assume comment records the entry state with respect to which the edit was made. If the program is run at a future time in a new environment, then debugging is simplified. The cause of a BELOW violation will now immediately be seen to be an incorrect assumption, and the corresponding repair is obvious -- insert code to satisfy the entry requirements described by the assumption. This illustrates the existence of levels of commentary between the model and the program, each layer being more specific, but also more closely tied to the particular code and runtime environment of the program.

Linear debugging greatly restricts the possibilities that must be considered to repair a violation. It is often successful and constitutes a powerful first attack on the problem of finding the proper edit; however, it is not infallible. Non-linear bugs due to unexpected interactions between main-steps would not be caught by this technique.

Figure 1.9 illustrates a non-linear bug. (INSIDE MOUTH HEAD) is violated but it cannot be repaired by adjusting the interface between these two parts (indicated in figure 1.9 by the dotted line OP) since the mouth is longer than the diameter of the head. The imperative semantics for fixing INSIDE recognize this. Consequently, MYCROFT resorts to the non-linear technique of modifying main-steps to repair a relation between parts. The imperative semantics suggest changing the size of one of the parts because this transformation does not affect the shape of the part and consequently will probably not introduce new

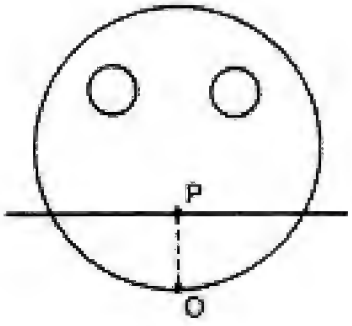


FIGURE 1.9

violations in properties describing the part. Advice is required from the user to know whether shrinking the mouth is to be preferred to expanding the head. Two more non-linear debugging techniques are discussed in the next two sections: one is based upon knowing the abstract form of plans, and the other uses domain-dependent theorems about global effects.

1.6 INSERTIONS

In programming, an interrupt is a break in normal processing for the purpose of servicing a surprise. Interrupts represent an important type of plan: they are a necessary problem solving strategy when a process must deal with unpredictable events. Typical situations where interrupts prove useful include servicing a dynamic display, and arbitrating the conflicting demands of a time sharing system. In the real world, biological creatures must use an interrupt style of processing to deal with dangers of their environment such as predators.

A very simple type of interrupt is one in which the program associated with the interrupt is performed for its side effects and is state-transparent, i.e. the machine is restored to its pre-interrupt

state before ordinary processing is resumed. As a result, the main process never notices the interruption. In the turtle world, an analogous type of organization is that of an inserted main-step (insertion). It naturally arises when the turtle, while accomplishing one part of a model (the interrupted main-step), assumes an appropriate entry state for another part (the insertion). An obvious planning strategy is to insert a sub-procedure at such a point in the execution of the interrupted main-step. Often, the insertion will be state-transparent: for turtles, this is achieved by restoring the heading, position and pen state. The insertion of the arms into the body by statement 30 of NAPOLEON is an example of a position- and pen- but not heading- transparent insertion.

Insertions do not share all of the properties of interrupts. For example, the insertion always occurs at a fixed point in the program rather than at some arbitrary and unpredictable point in time. Nor does the insertion alter the state of the main process as happens in an error handler. However, if one focusses on the planning process by which the user's code was written, then the insertion as an intervention in accomplishing a main-step does have the flavor of an interrupt.

The FINDPLAN module aids the debugger in a second way beyond just the generation of the plan. This is through the creation of caveat comments to warn the debugger of suspicious code that fails to satisfy expectations based on the abstract form of the plan. In particular, if FINDPLAN observes an insertion that is not transparent, then the following caveat is generated:

```
30 VEE  <- (caveat findplan (not (rotation-transparent insert))).
```

The non-transparent insertion may have been intentional, e.g. the preparation for the next piece of the interrupted main-step may have

been placed within the insertion. The user's program may have prepared for the next main-step within the insertion. Hence, FINDPLAN does not immediately attempt to correct the anomalous code. Only if subsequent debugging of some model violation confirms the caveat is the code corrected. There will often be many possible corrections for a particular model violation. The caveat is used to increase the plausibility of those edits that eliminate FINDPLAN's complaint. In this way, the abstract form of the plan helps to guide the debugging.

For NAPOLEON, analysis of (NOT (LINE BODY)) leads MYCROFT to consider (1) adding a rotation as statement 35 to align the second piece of the body with the first or (2) placing this rotation into VEE as the final statement. Ordinarily, linear debugging would prevent the latter as it does not respect the inviolability of main-steps. However, it is chosen here because of the corroborating complaint of FINDPLAN. The underlying cause of the bug is a main-step error (non-transparent insertion) rather than a preparatory-step failure. Thus, (DEBUG (LINE BODY)) produces:

70 RIGHT 45 <- (setup heading such-that (transparent vee))

1.7 GEOMETRIC KNOWLEDGE

Linearity, preparation and interrupts are general problem-solving strategies for organizing goals into programs. However, it is important to remember that domain-dependent knowledge must be available to a debugging system. The system must know the semantics of the primitives if it is to describe their effects.

The debugger must also have access to domain-dependent information to repair main-steps in which the sub-parts must satisfy certain global relationships. For example, TRICORN has the bug that the triangle is

not closed. Each main-step independently achieves a side but the sides do not have the proper global relationship. Debugging is simplified by the explicit statement in the model that:

(FOR-EACH ROTATION (= (DEGREES ROTATION) 120)).

But suppose the model imposed no constraints on the rotations. Then the design of the rotations would have to be deduced from such geometric knowledge as the fact that N equal vectors form a regular polygon if each rotation equals 360/N degrees.

The pieces of an interrupted-step such as the first side of TRICORN are not always separated by a state-transparent insert. (This would be a local interruption.) Instead, it is possible that more global knowledge is needed to understand the properties of the intervening code which justifies the expectation that the pieces will properly fit together. In TRICORN, the second piece (drawn by statement 70) must be collinear with the first (drawn by statement 10). The global property of the code which justifies this is that equal sides and 120 degree rotations results in closure. Thus, debugging violations of globally interrupted-steps requires domain-dependent knowledge.

Geometric knowledge does not replace the need for general debugging strategies: these are still very important to narrow the space of possible repair-points for correcting a given violation and to choose between alternative corrections. Section 4 discusses both types of knowledge in greater detail.

2. THE ANNOTATOR

Debugging is impossible without good description of a program's purpose and performance. MYCROFT begins with the program and a model describing its intended result. Two forms of additional commentary are then generated: Performance Annotation documents the effect of running the program while the Plan explains the intent. This commentary is organized as sets of assertions in a database, bound together into sequences representing what happened and why. Figure 2.1 shows part of the database generated to describe NAPOLEON. The nodes are organized so that the horizontal axis represents time and is used to answer such causal questions as what changes occurred to which state variables and which code was responsible for those changes. Similar data structures for describing programs are used by Fahlman [1973] and Sussman [1973].

The vertical axis represents teleological abstraction and explains the purpose of the code. Models fit into this descriptive framework as the highest level of abstraction. They describe the final goal without ties to specific plans or chronological performance. The next level is the plan, indicating the sub-goal organization for accomplishing the model. Finally, the teleology rests on a description of the actual performance of the turtle program when executed in a particular initial environment.

MYCROFT analyzes a program by first building a complete performance annotation and then applying the plan-finder to assign purposes to the code. Performance annotation is accomplished by running the user's turtle program in a "careful mode" which produces three kinds of description.

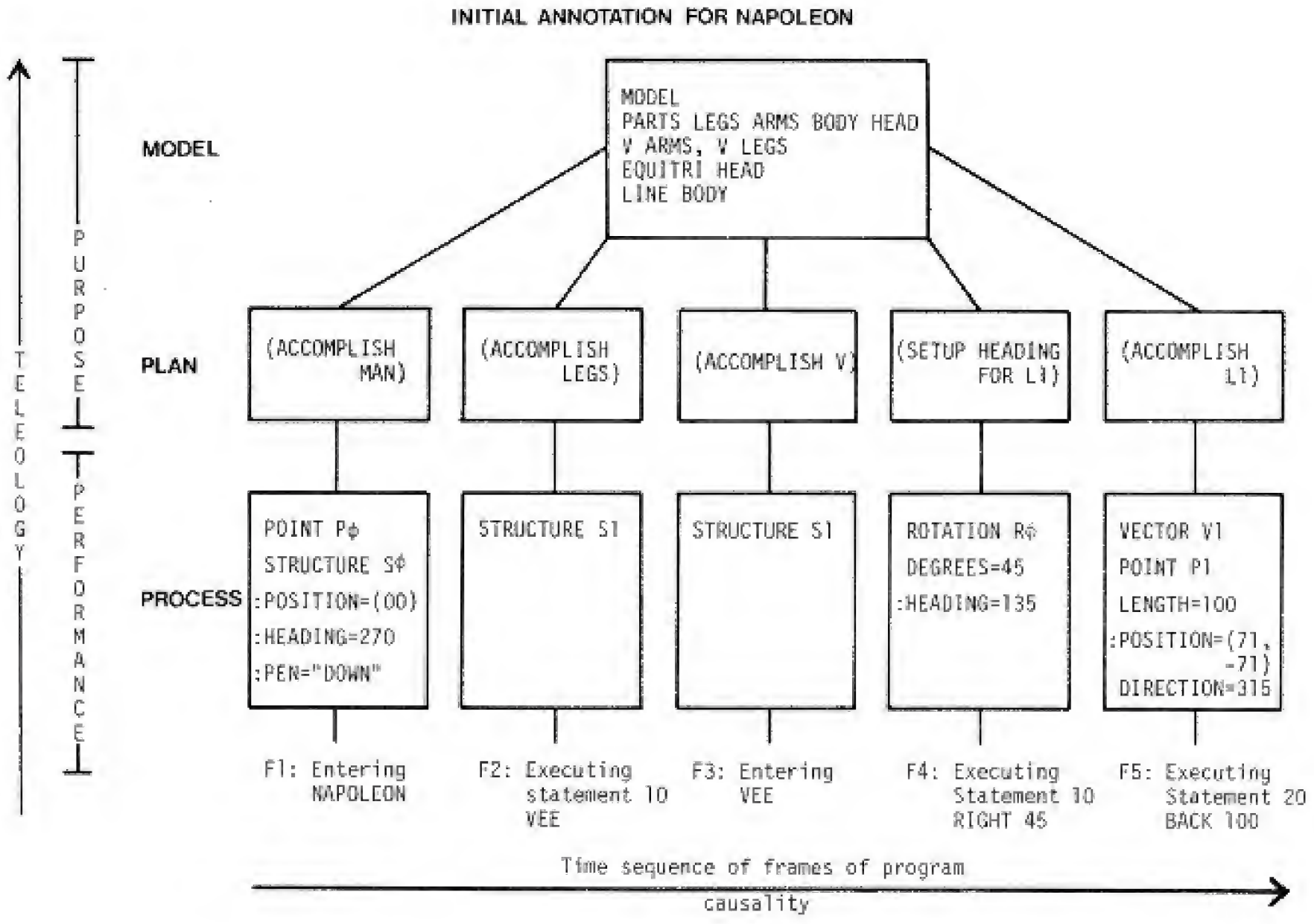

INITIAL ANNOTATION FOR NAPOLEON
TELEOLOGY
PURPOSE
PERFORMANCE
MODEL
PLAN
PROCESS
MODEL
PARTS LEGS ARMS BODY HEAD
V ARMS, V LEGS
EQUITRI HEAD
LINE BODY
(ACCOMPLISH MAN)
(ACCOMPLISH LEGS)
(ACCOMPLISH V)
(SETUP HEADING FOR L1)
(ACCOMPLISH L1)
POINT P$
STRUCTURE S$
:POSITION=(00)
:HEADING=270
:PEN="DOWN"
STRUCTURE S1
STRUCTURE S1
ROTATION R$
DEGREES=45
:HEADING=135
VECTOR V1
POINT P1
LENGTH=100
:POSITION=(71, -71)
DIRECTION=315
F1: Entering NAPOLEON
F2: Executing statement 10 VEE
F3: Entering VEE
F4: Executing Statement 10 RIGHT 45
F5: Executing Statement 20 BACK 100
Time sequence of frames of program causality

FIGURE 2.1

1. Process Annotation is a description of the output of the program. It consists of a record of the effects of executing each program statement. For turtles, this consists of the creation of vectors, vector structures, rotations and points.

2. Planning Advice suggests the segmentation of the program with respect to accomplishing the model on the basis of such criteria as global connections.

3. Debugging Advice describes suspicious code by caveat comments which aid in subsequent debugging.

Details of these three kinds of performance annotation are given below. The FINDPLAN algorithm is then described in section 3.

2.1 PROCESS ANNOTATION

Process annotation provides a description of the output of a program and its sub-procedures in terms of some language appropriate to the purpose for which the program was designed. For example, the performance annotation for an arithmetic program might be in terms of mathematical equations to be satisfied at various points in the computation [Floyd 1967]. For turtle programs, an obvious choice is to produce a Cartesian description of the picture drawn by the program. Annotation should reveal the basic effects of the code, free of vagaries of individual programming style. This would include knowing the description of a vector, regardless of whether the actual command is FORWARD, BACK or SETXY. (The last command moves the turtle to an absolute position on the screen.)

Annotation produces a sequence of frames. A frame is generated to describe the execution of each primitive and sub-procedure call. Each frame is a set of assertions specifying (1) any changes to the turtle's state and (2) the properties of any picture elements which have been created. The turtle's state consists of the values of the global variables :HEADING, :POSITION and :PEN. Picture elements (created as

side effects of executing turtle commands) are vectors, rotations, points and structures (vector sets drawn by recognizable code segments such as sub-procedures).

2.2 SEMANTICS FOR TURTLE PRIMITIVES

The process annotation is generated by imperative semantics associated with each turtle primitive. These semantics describe the performance of the turtle command.

```
SEMANTICS FOR (FORWARD :DISTANCE)       ;Draws a vector.

        (:VECTOR <-- (GENERATE-NAME 'V))
            ;All vertices, rotations, vectors and structures
            ;are given unique names to facilitate later debugging.
            ;If subsequent investigation reveals that the
            ;particular object has been given a label by
            ;the user, then the system name is replaced by the
            ;user's identifier.

    ;Describe the Vector in terms of its direction and length.

        (ASSERT (= (DIRECTION :VECTOR) :HEADING))
        (ASSERT (= (LENGTH :VECTOR) :DISTANCE))
        (ASSERT (= (VISIBILITY :VECTOR) <PENUP, PENDOWN, RETRACE>)

    ;Update the State of the Turtle

        (:POSITION <-- (FORWARD :DISTANCE))
                ;FORWARD :DISTANCE outputs coordinates of the new
                ;position.  Set the turtle state variable :POSITION
                ;to this new location of the turtle.

        (:POINT <-- (GENERATE-NAME 'P))
                ;If the coordinates are unique, bind :POINT to
                ;a new name for this position.  If not, use the
                ;old name for the position.  If a name already
                ;exists for this position, record the connections
                ;occurring at this point between :VECTOR and
                ;previous vectors.
```

```
SEMANTICS FOR (RIGHT :ANGLE)        ;Rotates the turtle.

        (:ROTATION <-- (GENERATE-NAME 'R))

    ;Describe the Rotation in terms of its vertex and degrees.

        (ASSERT (= (DEGREES :ROTATION) :ANGLE)
        (ASSERT (= (VERTEX :ROTATION) :POSITION)

    ;Update the State of the Turtle

        (:HEADING <-- (RIGHT :ANGLE))    ;RIGHT outputs the new heading.
```

At the level of the process, actual numerical values are determined for the above properties. Because these assertions depend upon the particular state of the initial environment, this is the most specific, least abstract level of commentary when compared with the model and plan.

2.3 PLAN-FINDING ADVICE

Although performance annotation does not examine the model, it can reveal clues to the grouping of the user's program into main- and preparatory-steps which aid in finding the plan.

1. Sub-procedures that draw visible sub-pictures are hypothesized to be main-steps that accomplish some model part.

2. Maximal sequences of "invisible" primitives such as (a) vectors drawn either by retracing or with the pen up, (b) rotations, and (c) PENUP commands are grouped together as possible preparatory-steps.

3. Maximal sequences of visible vector instructions plus any intervening rotations are grouped as possible main-steps.

4. Global connections suggest code boundaries. Thus, maximal sequences of visible vectors can be segmented on the basis of such connections.

This segmentation is tentative and may be revised in the light of later consideration of the model.

Suppose NAPOLEON was not subroutinized and, instead, the arms,

legs and head were open-coded (i.e. coded as in-line sequences of primitives rather than subroutinized). The above clues would be quite useful by utilizing the global connections between the body and limbs in the picture to suggest main-step boundaries.

2.4 DEBUGGING ADVICE

Oddities in the form of the program can create a suspicion of bugs. The annotator notices these violations using Rational Form Criteria which are sensitive to unexpected and apparently erroneous code. Caveat comments are generated describing these complaints.

Rational Form Criteria are based upon expectations of simple efficiency and consist of noting sequences of contiguous uses of the same primitive, such as FORWARD, RIGHT or PENUP. The annotator considers the code to be odd: why didn't the user simply coalesce them into a single call with a larger input or, in the case of PENUP, include only the first instruction? The answer may be that the user has forgotten to insert additional instructions. An example would be where the user had forgotten to insert several RIGHT commands into a sequence of FORWARD instructions. A caveat stating that code may be missing is placed between each pair of elements in the sequence of FORWARD's. A violation of rational form occurs in the following triangle procedure because the user has forgotten the first rotation.

```
TO TRI
10 FORWARD 100 <- (caveat annotator RATIONAL-FORM-VIOLATION
                           (sequential-primitive 10 30))
30 FORWARD 100
40 RIGHT 120
50 FORWARD 100
END
```

An edit that inserts a rotation into such a sequence of FORWARD instructions would eliminate the rational form violation and therefore

be preferred in competition with other corrections which do not explain the annotator's complaint. If the debugger corrects the program by eliminating the annotation caveat, then the underlying cause of the error is considered to be "Missing Code".

3. THE PLAN-FINDER

After performance annotation, the next step in describing the program is to find the plan. The strategy is to attempt initially to find a linear plan, i.e. to match model parts with modular main-steps and relations between model parts with preparatory-steps. This approach serves to limit the search space, but it is not adequate to recognize interrupted main-steps and insertions. These "non-linearities" are suggested by suspicions about the cause of violations implied by the conjectured linear plan. These suspicions are that the cause of the violation is not an error in the user's program but a mistake in the plan-finder's linear interpretation of the plan. If additional evidence confirms the suspicion, the plan-finder corrects its linear analysis and finds the correct global or insertion type of plan. This approach of first pursuing a linear interpretation and only 'debugging' this approach in response to anomalies is a powerful reasoning mechanism for searching complex spaces. As was noted in section 1, the debugger uses a similar analysis to simplify finding the proper repairs.

Plan-finding obtains some guidance from the picture and some from the program. The picture supplies such clues as:

(a) global connections which suggest sub-picture boundaries;

(b) retracing which suggests inserts;

and (c) violations of model statements which are then used both as plausibility criteria (to distinguish between alternative interpretations) and to generate suspicion demons (which look for non-linear planning structures).

The program supplies quite different clues about intent. This includes:

(a) sub-procedure structure which aids in recognizing main-steps;

and (b) the order in which the picture is drawn which, when combined with program-writing criteria, suggests the order in which the

model parts are accomplished.

3.1 PLAN-FINDING AS SEARCH

Finding the plan can be conceptualized as a search of a space of "partial plans". The search begins with the model, the program and the performance annotation. A partial plan is an explanation of some fraction of the model in terms of the program. Given a partial plan, its daughters are the result of generating alternative explanations for one of the remaining unassigned model parts. A terminal node is reached when all of the model parts have been explained and a complete plan is a path from the root to a terminal node, wherein an explanation is provided for how each model part is achieved.

A partial plan consists of PURPOSE comments which assign model predicates to code, unassigned model parts, expectations, the implied partial interpretation, and demons.

PURPOSES - These are the basic statements of a plan and appear as "<-" commentary in the NAPOLEON procedures. Five kinds of purposes are generated by FINDPLAN: accomplish, insert, setup, cleanup and retrace.

UNASSIGNED MODEL PARTS - The model specifies a list of parts. These are either primitive picture objects (vectors or rotations) or sub-models. An unassigned part is one without a PURPOSE statement indicating how it is to be accomplished.

EXPECTATIONS - These are predictions of which part is expected to be accomplished by the next main-step. They are based on applying program-writing criteria of efficiency and simplicity to the model. See the discussion of Analysis by Synthesis in the next section.

PARTIAL INTERPRETATION - Model predicates can be evaluated by ordinary Cartesian geometry using the binding of model parts to code (which the plan implies) and an annotated description of the code's effects. A partial interpretation consists of those model predicates whose truth value is known given the current partial interpretation.

DEMONS - Demons are used to explain subsequent code in such a way that violations in the partial interpretation are eliminated. The elimination results from debugging the system's linear analysis and

recognizing the existence of an interrupted or inserted main-step.

The partial plan is complete when all of the unassigned parts are explained by PURPOSES. Debugging is fixing the violations of the resulting complete interpretation.

3.2 LINEAR PLAN SPACE

The search is neither a standard breadth nor depth first exploration of the space. Instead, the system initially assumes a linear structure to the user's plan, looking to assign the parts to sequential code segments. The possibility that a part is being accomplished by disjoint segments of code or by insertions is not considered. This greatly constrains the search space. Branching, however, is not eliminated: for a given program, more than one linear plan will usually be possible. To choose among the alternatives in this linear plan space, several plausibility criteria are used.

1. (Advice) The first is to take advantage of user, annotator or debugger advice to initialize the partial plan space. Annotator advice originates in noticing (1) sub-procedures that have been previously associated with a model and (2) open-coded sequences identified as having a common purpose on the basis of non-model clues like penstate changes and retracing. (See section 2.3.) The first produces PURPOSE assertions which form the initial partial plan: the second SUGGESTIONS which have the effect of causing open-coded sequences to be treated as sub-procedures. Debugging advice is in the form of a request that the plan-finder supply a new plan that does not make certain hypotheses about the program. This interaction arises when the debugger finds all editing strategies for the current plan implausible.

2. (Analysis by Synthesis) Another method is to consider the model from the point of view of program writing. This leads to two forms of advice. The first is to assign sub-procedures to model parts if possible (on the grounds that the model parts constitute a likely plan for breaking the picture into sub-goals). The second is to generate expectations for the order in which the parts are to be accomplished. This is done by observing transitive sequences of such predicates as BELOW and CONNECTED in the model. The heuristic is that that these sequences represent the probable order in which the parts are accomplished, thereby minimizing retracing.

3. (Static Evaluation Function) A third method is a plausibility estimate of partial plans. This estimate is simply the number of satisfied model statements and expections minus the number of violated model statements and expectations. If the program is bug free and the plan is correct, then the plausibility number will be maximal. At any instant in time, only those plans with the highest plausibility number are explored. After analyzing a statement of code, the plausibility number is recomputed and the active plans are rechosen. Inactive plans are "hung" and are not resumed unless their active brethren become less plausible.

3.3 FINDING THE PLAN FOR STICKMAN

As an example, let us consider the problem of finding the plan for NAPOLEON. Recall that the procedure is:

```
TO NAPOLEON                ;See figure 1.1
10 VEE
20 FORWARD 100
30 VEE
40 FORWARD 100
50 LEFT 90
60 TRICORN
END
```

We shall assume that the VEE sub-procedure has been previously annotated and associated with the V model but that TRICORN and NAPOLEON have just been defined and their purpose is unknown. By considering sub-procedures as candidates for accomplishing model parts (analysis by synthesis), TRICORN is bound to the EQUITRI model. The result is two possible initial partial plans. These are:

```
PARTIAL.PLAN.1:                    PARTIAL.PLAN.2:
10 VEE <- (accomplish legs)        10 VEE <- (accomplish arms)
30 VEE <- (accomplish arms)        30 VEE <- (accomplish legs)
60 TRICORN <- (accomplish head)    60 TRICORN <- (accomplish head)
```

Further constraints are imposed by FINDPLAN's program-writing expectations. On the basis of BELOW, FINDPLAN expects:

(accomplish legs) <-> (accomplish arms) <-> (accomplish head)

The double arrow indicates that the sequence may happen in either forward or reverse order. On the basis of connectivity, the

expectations are:

(accomplish legs) <-> (accomplish body) <-> (accomplish head)

Taken together, the result is that statement 10 is believed to accomplish the LEGS and statement 30 the ARMS. Thus, PARTIAL.PLAN.1 is preferred.

The code of the program is then considered statement by statement. Statement 20 draws a vector and is therefore believed to be the BODY. It might be only a piece of the body but this is not pursued until the linear assumption that the body is accomplished by a modular main-step is rejected.

Statements 30 and 60 have already been assigned to the arms and head, respectively. As a result, all of the model parts have been assigned but statement 40 remains unexplained. FINDPLAN consequently backtracks and interprets statement 20 as only piece of the body. A demon is created for recognizing the body's completion and plan-finding recommences at statement 30. Statement 40 satisfies this demon since it draws a vector that begins at the endpoint of the first piece of the body. The result is that it is considered (piece 2 body). Thus, with almost no search, the plan for NAPOLEON is correctly deduced.

```
TO NAPOLEON                    <- (accomplish man)
10 VEE                         <- (accomplish legs)
20 FORWARD 100                 <- (accomplish (piece 1 body))
30 VEE                         <- (insert arms body)
40 FORWARD 100                 <- (accomplish (piece 2 body))
50 LEFT 90                     <- (setup heading)
60 TRICORN                     <- (accomplish head)
END
```

3.4 NON-LINEAR PLANS AND SELF CRITICISM

This section explains how interrupted and inserted main-steps are recognized. When FINDPLAN binds an unassigned model part M to a segment of code C and the resulting interpretation implies model violations, there are three possible explanations:

1. The code is in error: a bug has been discovered.

2. C is not intended to accomplish M. Choose another interpretation for C.

3. C accomplishes only a PIECE of M. The remainder of M is achieved in pieces.

Possibility 1 requires no special action by FINDPLAN: the violation will eventually be passed to DEBUG for correction. Possibility 2 requires that the a different linear plan be chosen. This will occur if the current linear plan becomes less plausible than alternative linear interpretations when compared in terms of the static plausibility function described earlier. Possibility 3, however, represents an error in the plan-finder's linear analysis of the program. Hence, to take account of possibility 3, demons are generated. These demons are looking for better interpretations than the current linear plan (i.e. interpretations which do not imply as many violations). The following paragraphs describe the creation of such a demon in the plan-finding process for TRICORN.

Suppose FINDPLAN has just decided that statement C achieves model part M and that this results in a violation because M is too small. FINDPLAN suspects that M may be being accomplished in pieces. A COMPLETION demon is created looking for subsequent code CC which would eliminate the violation if CC is interpreted as another PIECE of M. If such code is found, the action of the demon is to edit the original partial plan so that M is now considered as being achieved by an

interrupted main-step. If the code between the pieces of the main-step returns the turtle to the exit state of the first piece, then it is interpreted as being an insertion. COMPLETION demons are also created when a vector is too short to accomplish an intended connection. An example occurs in the linear interpretation of TRICORN shown below:

```
TO TRICORN       ;Incorrect linear plan initially deduced.
10 FORWARD 50    <- (accomplish (side 1))
20 RIGHT 120     <- (accomplish (rotation 1))
30 FORWARD 100   <- (accomplish (side 2))

  ;At this point in the plan-finding process, the violation
  ;of unequal sides occurs.  A COMPLETION demon is created
  ;that is looking for a vector of length 50 that could be
  ;interpreted as the remainder of (side 1).

40 RIGHT 120     <- (accomplish (rotation 2))
50 FORWARD 100   <- (accomplish (side 3))

   ;Here the violation of (side 1) not being connected to
   ;(side 3) occurs.  A second COMPLETION demon is created
   ;that is looking for another PIECE of (side 1) that connects
   ;to (side 3).

60 RIGHT 120     <- (accomplish (rotation 3))
70 FORWARD 50    <- (accomplish 7)
END
```

Both of the COMPLETION demons are triggered by statement 70. The result is that statement 10 is reinterpreted to accomplish only (piece 1 (side 1)) and statement 70 is assigned the purpose of accomplishing (piece 2 (side 1)). This produces the correct plan. (Other demons are created in the plan-finding process for TRICORN. However, they are never triggered and are therefore not mentioned.)

3.5 SUMMARY OF THE PLAN-FINDER

The algorithm for plan-finding performs well when:

(1) The user supplies advice in the form of a partial plan;
(2) The procedure has subroutines;
(3) The procedure has few bugs.

If the program is not subroutinized and is full of bugs, the search grows unmanageable and difficulties arise in selecting the most plausible candidate. This performance is quite reasonable in the sense that similar statements are true of a human problem solver investigating a strange program.

4. THE DEBUGGER

4.1 MODEL VIOLATIONS

The monitor is designed to debug model violations. These are recognized by the INTERPRET module (see again figure 1.6) which compares the output of a syntactically and semantically correct turtle program (i.e. a program that is able to run to completion without requesting any illegal computations) to the description of intent provided by its picture model, using the plan to bind sub-pictures to model parts. The result is a list of violated model predicates. The program is considered correct when all of these violations have been eliminated.

Correcting model violations is accomplished by using two types of procedural knowledge: (1) a collection of general debugging strategies for repairing programs and (2) directions for fixing particular geometric and logical predicates. Because overall guidance is derived from the model, we shall call this type of analysis model-driven debugging.

4.2 DEBUGGING AS SEARCH

A debugging strategy is a sequence of editing commands whose effect is to modify the program so that it satisfies its model. There are generally multiple debugging strategies for correcting a given set of violations. These alternative debugging strategies arise from choice of the repair-points at which the corrections are to be made as well as of the exact meaning that the user intended.

To clarify the issues which arise in selecting the best debugging sequence, it is useful to conceptualize the problem in terms of a search metaphor. The space is that of all possible debugging

strategies for correcting the program. Each node is a set of model violations: the origin of the space is the initial output of INTERPRET. An arc is an edit which which leads to a node containing the new (and presumably smaller) set of violations which are produced by the patched code. Branching occurs for each possible patch for correcting a violation. A path through the space constitutes a series of edits that transform the program to an acceptable form.

Recognizing the existence of multiple possibilities for correcting a program, it is appropriate to ask what knowledge is used to:

(1) choose the next model violation to be debugged?
(2) generate the possible corrections for that violation?
(3) choose the most plausible correction?

The following sections answer these questions. Ordering Criteria are introduced for choosing the sequence in which the violations are debugged. A linear approach curtails the number of possible edit points which are initially considered. The imperative semantics of the model predicates are used to generate possible corrections. Plausibility criteria are designed for selecting among alternative debugging strategies.

4.3 ORDERING MULTIPLE VIOLATIONS

Multiple bugs are difficult to fix. Guidelines are required to order the sequence in which the violations are debugged. These guidelines reflect an understanding of dependency relationships between violations, thereby serving to minimize the unfortunate occurrence of a correction undoing previous repairs or introducing new violations. The ordering is done on the basis of preferring to repair:

(1) bugs in properties of model parts before bugs in relations between model parts;

(2) bugs in intrinsic properties (or relations) before bugs in extrinsic properties (or relations);

and (3) bugs occurring earliest in the temporal sequence of execution.

The following paragraphs describe these criteria and explain their rationale.

4.3.1 Debug Properties Before Relations

The system debugs violations of properties of model parts before repairing violations of relations between model parts. This is based on the important heuristic of first having a successful theory of the parts before attempting an explanation of their interactions. This is more than good style. The behavior of the interfaces is designed relative to the entry-exit states of the code for the main-steps accomplishing the parts. To determine the specific state changes to be made at an interface, the performance of adjacent main-steps must be established. Thus the code for sub-pictures must be fixed prior to deciding on the proper edits to the preparatory-steps.

Properties of individual model parts include unary model primitives (e.g. VERTICAL, HORIZONTAL and LINE) as well as user-defined sub-models (e.g. EQUITRI and V). The most common relations between model parts are predicates such as ABOVE, BELOW, and CONNECTED.

4.3.2 Debug Intrinsic Before Extrinsic Predicates

The idea behind the next ordering criteria is to estimate the range of possible locations in the program at which the repair might be made for each violation. The heuristic is then to fix those violations

of most-limited scope first; both because they are easiest and because of dependency relationships.

Let the scope of a violation be the code between the repair-point and the manifestation-point. For a property (P M), M a model part, the manifestation-point is the location in the program at which M is completed and the truth of the statement (P M) can be evaluated. The repair-point is the location in the program at which the edit is eventually made to correct the violation. For a relation (R M N), the manifestation-point is the location in the program at which both M and N have been completed and the relation R can be evaluated.

This criterion would be pointless if there were no way to estimate the scope of a violation before entering into the details of debugging. However, this is not the case. One method for estimating the scope of a violation is to know whether the property of relation is intrinsic to the responsible code.

A property (P A) is intrinsic to the code for A if it is independent of preceding code and entirely due to the main-step for A. Similarly, the relation (R A B) is intrinsic if it is independent of code preceding A, assuming that A is achieved before B. Repair is simplified by fixing intrinsic predicates before extrinsic ones since (1) for intrinsic violations, the possible repair-points are easier to find since they cannot occur prior to the code for A, and (2) the proper corrections for extrinsic predicates depends upon the the code being intrinsically correct.

In the world of turtle geometry, intrinsic errors are distinguished by being independent of the frame of reference: they cannot be corrected by translating or rotating the picture. This is because in the simplified environment of fixed-instruction turtle

programs, code groups draw rigid bodies. The initial interface of a code group has the effect of establishing the origin and orientation of the sub-picture but does not affect the local relations among vectors. Topological predicates (invariant under transformations that preserve connectivity) and geometric predicates (invariant under translation and rotation) are independent of the frame of reference and therefore yield intrinsic violations. Bugs in the following model primitives are always intrinsic to the code group to which they refer: OVERLAP, INSIDE, OUTSIDE, PARALLEL and CONNECTED.

Extrinsic errors are those affected by the initial environment in which the code group is executed. The initial environment consists of the bindings of the turtle state variables -- :HEADING, :POSITION and :PEN. These variables control the orientation, origin and visibility of the sub-picture as well as its relation to previously drawn parts of the picture. Model predicates which depend on the initial state are VERTICAL, HORIZONTAL, BELOW, and ABOVE.

Debugging intrinsic violations first tends to establish the proper connections at interfaces. Debugging extrinsic relations like ABOVE then becomes simply a matter of establishing the proper heading at interfaces.

In the turtle world, the distinction between intrinsic and extrinsic predicates is particularly easy to make; however, it remains a useful debugging distinction in other domains. If a property of a program is due to some local data structure (such as a bound variable) or local control structure (such as a loop) and is independent of the preceding code, then it is intrinsic and worth debugging in private before extrinsic properties (whose causes are less easy to isolate) are repaired.

4.3.3 NAPOLEON's Violations

The following list of violations for NAPOLEON is ordered by the above criteria:

```
(Violations of Properties of Parts of NAPOLEON)
  (An Intrinsic Violation -- Manifested in Private)
    (NOT (EQUITRI TRICORN))

  (An Extrinsic Violation -- Not Manifested in Private)
    (NOT (LINE BODY))

(Violations of Relations between Parts of NAPOLEON)
  (Temporal Order -- {legs, arms} accomplished before {arms, head})
    (NOT (BELOW LEGS ARMS))
    (NOT (BELOW ARMS HEAD))
```

4.4 FINDING THE PROPER REPAIR-POINT

For each violation, DEBUG must find the proper repair-point in the program at which to insert the correction. Of course, the debugger knows that the repair-point cannot follow the code for the parts mentioned in the violation but this is hardly a sufficient constraint. Consequently, DEBUG uses two heuristics--Private and Linear Debugging--to limit the possible locations for the correction.

4.4.1 Private Debugging

An initial heuristic for constraining the possible repair-points for a violated property is to limit consideration to the code directly responsible for the model part in question. This is done by running the responsible code independently of the larger procedure of which it is a part. Specifically, the responsible code is executed with the turtle started at the entry state. The violated properties will be manifested in this private environment if the main-step is modular. However, if there is intervening code, i.e. the main-step is interrupted, then the

linear assumption that the cause is intrinsic to the responsible code and not due to interactions may be wrong.

If the violation is manifest, the code group is then debugged in this simplified context, free of the effects of the remainder of the original program. Private debugging is used to repair the three incorrect rotations of TRICORN. There are no complications when the edited sub-procedure is rejoined to the NAPOLEON super-procedure.

The relationship between the picture drawn in private and in public is simple for fixed-instruction turtle programs since the picture is a rigid body and only its orientation and origin is affected by the initial environment. For more complex programs, difficulty occurs in finding a representative private environment and further research is necessary. This is similar to the problem of diagram generation in geometry theorem proving and to the problem of case analysis in automatic program verification.

The private repair may make assumptions about the entry state to the code. If this happens, it will be reflected in ASSUME comments regarding the entry state to the main-step. When run again in the real context, any conflicts between assumptions made in private about the initial environment and the actual entry state are themselves debugged. This is accomplished by adding code to accomplish the assumptions in the super-procedure or, if this proves impossible without causing additional violations, backtracking and attempting an alternative correction in private.

An example of this would occur if the model for NAPOLEON had declared that the body must be vertical. Debugging the body (statements 20 and 40) in private would result in the assumption being generated that the entry heading must be 0 or 180 degrees. The code for the body

is then reconsidered in the context of the NAPOLEON super-procedure. The actual entry state to statement 20 does not have :HEADING equal to 0 or 180 degrees. Consequently, the debugger now attempts to add a rotation at some preceding point in the program to achieve this entry state. This addition will most likely occur immediately prior to statement 20 or, perhaps, as the initial setup to the NAPOLEON program. The debugger chooses whether to prefer 0 or 180, and at which repair-point, on the basis of side effects, minimal change to the user's program and planning caveats. This set of plausibility criteria is described in section 4.7.

The system also checks for bad side-effects on code following the edited sub-group due to a new exit state for the edited code. A cleanup step may be needed to eliminate undesirable consequences of the private repairs. The modified main-step may violate protection or assumption commentary generated by other edits. If so, the standard practice is to either (1) modify the offended edit in light of the new structure for the main-step or (2) backtrack and correcting the main-step in private in some alternative way. See section 4.6 for details on the protection mechanism.

Occasionally, when the code is run in private, the violation does not occur. This happens because the main-step is not modular and the violation is due to code appearing between pieces of an interrupted main-step. Private debugging remains useful, however, because it clearly indicates that the cause of the error is in the intervening code. (NOT (LINE BODY)) is an example: the body when run in private is indeed a line. The bug is in the effect of the inserted VEE on the heading of the second vector.

Private debugging is also used to correct intrinsic violations

of relations. Recall that the definition of an intrinsic relation is that it is entirely due to the code between the model parts mentioned in the relation. Hence, the repair-point must occur there. The same precautions required when the code is rejoined to the super-procedure--i.e. satisfying assumptions, and possibly cleaning up--must be taken. Outside the turtle world where it may not be so easy to decide if a relation is intrinsic, private debugging can still be attempted. Just as for properties, if the violation does not appear in private, then it is known that it is not intrinsic and the system can look for causes in preceding code.

4.4.2 Linear Debugging of Relations

Linear Debugging is a technique for limiting the possible repair-points for correcting violated relations of both the intrinsic and extrinsic kind. It is based upon the assumption that DEBUG has already privately repaired the main-steps to satisfy their properties. The linear debugging technique is to consider editing corrections only at preparatory-steps and not internal to the code for the main-steps. Main-steps are treated as inviolate black-boxes: their contents need neither be known nor changed. This is based upon the assumption that the main-steps are independent and that the only corrections necessary to repair relations is to make adjustments at interfaces. This was the technique used to debug (BELOW LEGS ARMS). DEBUG limited the search for the proper edit by not considering the addition of a rotation to the interior of the VEE sub-procedure. Instead, it restricted itself to an analysis of possible corrections at the level of the NAPOLEON super-procedure.

Linear debugging fails when the underlying cause of the

violation is due to the code for one of the parts. In such a case, it is necessary to remove the restriction against modifying main-steps. An example where this occurs was shown in figure 1.9. The violation of the mouth not being inside the head is caused by the size of the mouth, not by the interface.

4.5 IMPERATIVE KNOWLEDGE

How is the set of possible edits for repairing a violation generated? The answer lies in the use of procedural knowledge associated with the model primitives which provides direction on how to make the predicate true. The system has imperative knowledge for logical primitives like equality and conjunction as well as for geometric primitives appropriate to the turtle world. This imperative knowledge outputs a set of possible edits whose effect is to eliminate the violation.

In the NAPOLEON example, (NOT (EQUITRI TRICORN)) is a violation of a user-model. Such violations are fixed by recursive entry to the debugger and analyzing the code for the model in private. Such recursion ultimately reduces the debugging to fixing violations of model primitives.

4.5.1 Imperative Knowledge for Geometric Primitives

The following discussion describes in a simplified way the imperative knowledge associated with several of the model primitives. Let X and Y be vectors and assume that X is accomplished before Y.

(LINE X Y) <=> (AND (PARALLEL X Y) (CONNECTED X Y))

> The imperative semantics for AND directs debug to establish the two relations of PARALLEL and CONNECTED. These are defined below.

(PARALLEL X Y) <=> (= (DIRECTION A) (DIRECTION B) (MOD 180))

The annotator records the DIRECTION of vectors. The repair is to insert rotations between the code for X and the code for Y so that the direction of Y becomes equal to the direction of X (mod 180).

(VERTICAL X) <=> (OR (= (DIRECTION X) 0) (= (DIRECTION X) 180))

Alter preceding rotations so as to make the direction of X 0 or 180.

(CONNECTED X Y)

Choose a connection point on X (P1) and a connection point on Y (P2). The connection point is sometimes specified in the model: for example, the user may have indicated that it should occur (AT (MIDDLE (SIDE ...))). Then compute the vector V from P1 to P2. The edit is to add code for V into an interface between X and Y. This will have the effect of translating Y so that P1 is moved to coincide with P2.

If the exact position is unknown, deduce it from constraints such as preferring to effect the code in minimal ways. This is done by manipulating individually the length and angle inputs to translation and rotation interface steps (occurring between the code for X and the code for Y) and observing if X and Y intersect as a result. Branch in considering alternative allowable connection positions.

(ABOVE X Y) - (similar technique for BELOW, RIGHT-OF, LEFT-OF)

To compute the required correction for a given interface: assume that the figure has already been debugged to be topologically correct--e.g. all of the connections are correct. This implies that the only degree of freedom in interfaces is the heading.

In considering a given interface, find the range of headings which satisfy the predicate. The range is determined by first finding the heading of most restrictive meaning of ABOVE -- CENTERED-ABOVE wherein the center of gravity of X is directly above Y. Then relax this heading to find the maximum range in which less restrictive meanings of the predicate--COMPLETELY-ABOVE and PARTLY-ABOVE--remain true. To select a specific heading to actually edit into the code, choose the value that satisfies the most restrictive meaning of ABOVE. If there is still a range of possible headings, use the average value. Record the range considered in case later debugging results in conflicts and another heading must be chosen.

4.5.2 The Rigid Body Theorem

Fixed-instruction turtle programs draw rigid bodies, i.e. the only effect of the initial runtime environment is to alter the visibility, origin or orientation of the frame of reference. This

theorem simplifies the generation of possible repair edits by allowing computation of the required rotation for HORIZONTAL, VERTICAL and PARALLEL to be made only once, independently of the point in the code at which the edit is to be added. This is useful since there are usually many points at which patching the code must be considered to fix these violations.

For example, suppose the side of a triangle is to be made horizontal. The required rotation is computed for the side. However, if the edit is made immediately prior to the code for the side, the triangle shape will be destroyed. The rotation, however, can be added to preceding code, rotating all subsequent vectors the same amount and consequently still making the side horizontal.

In general, if the correction is a rotation of the frame of reference, the edit can be added anywhere prior to the code group to be rotated. If the rotation is to change the relation between two sub-pictures, then it can often occur anywhere in the code occuring between the main-steps which accomplish the sub-pictures.

4.5.3 Imperative Knowledge of Logical Predicates

The general advice for fixing (= (P A) (P B)) is to use the imperative semantics for property P to either make (P A) equal to (P B) or vice versa. For the simple case of fixed-instruction turtle programs, the change is usually made to A or B on the basis of which occurs last. This is preferred because of the rigid body nature of sub-pictures. For example, suppose A occurs before B. Then adding RIGHT :ANGLE before A rotates A but it also rotates B. An opposite rotation must be added after A if B is not to be affected by the first edit. Thus, fixing the sub-picture which occurs first commits the system to

two changes of the program. Of course, editing the code before B may also require a cleanup because of bad side effects but this is not inevitable as it is in the first case. This preference is reflected in the general debugging criteria of avoiding conflicts, minimizing change to the user's program and preferring beneficial side effects.

Thus, fixing equality consists of:

> General Knowledge: Either A or B can be fixed. Prefer to alter the unprotected element (section 4.6).
>
> Domain-Dependent Knowledge: Imperative semantics are provided for relating primitives to their effects. These semantics are used by the annotator to document the effect of a statement of code, and by the debugger to add the correct code to achieve a desired effect. For example, to alter the direction of a vector, the annotation semantics for FORWARD (section 2.2) indicate that the DIRECTION property of vectors is equal to the current heading. The annotation semantics for RIGHT indicate that :HEADING is incremented by :ANGLE following execution of "RIGHT :ANGLE". The conclusion drawn by the debugger, then, is that either "RIGHT :ANGLE" is needed to fix the direction of B or "RIGHT -:ANGLE" is needed to fix the direction of A, where :ANGLE equals the difference between the desired direction and the actual direction.

To fix (AND C1 C2 ...), correct all of the conjuncts. Order the debugging attack on the basis of the same criteria used to order the initial set of violations. Correct properties of main-steps before correcting relations between main-steps. Correct intrinsic before extrinic predicates. Debug a given group of conjuncts at the same level (with respect to the preceding criteria) in temporal order.

See [Goldstein 1974] for a description of imperative semantics for other model primitives such as INSIDE, OUTSIDE, OVERLAP, OR, NOT and FOR-ALL.

4.6 ASSUMPTION AND PROTECTION

DEBUG generates assumption and protection commentary associated with each repair to aid in resolving difficulties where an edit causes

new violations or undoes the effects of some previous edit. Assumptions about the entry state at the repair-point describe expectations on which the imperative semantics based their analysis. Protection commentary guards the code from the repair-point to the manifestation-point (the place in the code at which the sub-pictures referred to by the violated model predicate were completed), again because the details of the repair depend upon the state manipulations of the code between the edit and the manifestation-point. Protection is introduced by Sussman in the context of debugging blocks world programs [Sussman 1973].

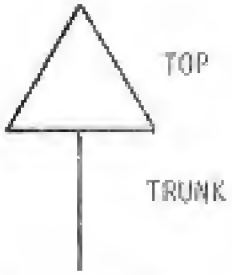
A simple example arises for the following tree program:

```
MODEL TREE       ;See figure 4.1.
M1 PARTS TOP TRUNK
M2 LINE TRUNK
M3 EQUITRI TOP
M4 VERTICAL TRUNK
M5 COMPLETELY-BELOW TRUNK TOP
M6 CONNECTED TOP TRUNK
M7 HORIZONTAL (BOTTOM (SIDE TOP))
END


TO TREE4          <- (accomplish tree)
10 TRIANGLE       <- (accomplish top)
20 RIGHT 60       <- (setup heading such-that
                        (overlap (interface statement 30) (side 3 top)))
30 FORWARD 50     <- (retrace (side 3 top))
40 RIGHT 45       <- (setup heading for trunk)
50 FORWARD 100    <- (accomplish trunk)
END

TO TRIANGLE       <- (accomplish equitri)
10 FORWARD 100    <- (accomplish (side 1 triangle))
20 RIGHT 120      <- (accomplish (rotation 1 triangle))
30 FORWARD 100    <- (accomplish (side 2 triangle))
40 RIGHT 120      <- (accomplish (rotation 2 triangle))
50 FORWARD 100    <- (accomplish (side 3 triangle))
                     (cleanup position)
60 RIGHT 120      <- (accomplish (rotation 3 triangle))
                     (cleanup heading)
END
```

See figure 4.2 for the picture drawn by TREE4 with the turtle starting at the center of the screen and with a heading of zero degrees.



Intended TREE

FIGURE 4.1

TREE 4
VERSION 1
Slanted Base and Trunk

FIGURE 4.2

TREE 4
VERSION 2
Base Made Horizontal

FIGURE 4.3

TREE 4
VERSION 3
Trunk Made Vertical

FIGURE 4.4

Debugging the base of the TOP to be horizontal results in the addition of statement 5 to rotate the triangle so that the necessary orientation is established. This produces figure 4.3.

5 RIGHT 30 <- (setup heading such-that (horizontal (side 3 top)))

Debugging the TRUNK to be vertical by modifying the initial setup, however, undoes this correction (figure 4.4).

3 RIGHT 45 <- (setup heading such-that (vertical trunk))

The solution is for the initial correction of (HORIZONTAL (SIDE 3 TOP)) to include commentary explaining its purpose, scope and assumptions. Specifically, this commentary is:

1. an assumption that the entry state to statement 5 is :HEADING=0:
   (ASSUME (TREE4 STATEMENT 5) (= :HEADING 0)).

2. a protection to any modifications of :HEADING from statement 5, the repair-point, to statement 50 of TRIANGLE, the manifestation-point of the error:
   (PROTECT :HEADING UNTIL (TRIANGLE STATEMENT 50)).
   Statement 50 is the manifestation-point of the error since it accomplishes (side 3) and INTERPRET is then able to recognize that a violation exists--the base of the triangle is not horizontal.

These comments force the debugger to prefer the alternative repair strategy of making the trunk vertical by editing the rotation of statement 40 to be RIGHT 90.

A second use of this commentary, in addition to preventing conflicts between edits, is to simplify debugging the procedure if it is ever run in a new environment. Unsatisfactory initial state values are immediately noticed by the assumption commentary. For example, if statement 5 of TREE4 contains the assumption that the entry heading should be 0, then being run in any other environment will generate a violation. This violation then directs the debugging.

> Thus, previous debugging sessions produce commentary whose specificity eliminates complex questions of responsibility and interpretion. The system has, in effect, generated the snapshots of performance which Naur and Floyd utilize to verify

programs [Floyd 1967, Naur 1967].

The assumption comment is passed to the debugger as an instruction and the result is that code is added prior to statement 5 which converts the heading to the desired value.

Often a protection conflict can be resolved. The debugger is simply recalled to achieve the edit which gave rise to the protection, taking into consideration the new entry or exit state requirements. This second call to the debugger involves less effort than the first. The commentary from the first remains and indicates the desired Cartesian state to be achieved at the manifestation-point. If the second edit succeeds without causing unfixable violations as side effects, then the system has patched its own edit and need not reject the basic form of its current analysis.

4.7 DECIDING BETWEEN ALTERNATIVE DEBUGGING STRATEGIES

More than one debugging strategy is usually available to fix a given violation. The strategies differ with respect to their estimate of the failure point and with respect to the type of correction they apply to fix a given model violation. For example, the imperative semantics for BELOW indicate the desired direction but allow the correction to be added into any prior interface. In NAPOLEON, the arms can be made above the legs by adding the appropriate rotation to the beginning of the NAPOLEON procedure or immediately following statement 10, the code for the LEGS. The preferred debugging strategy is the one that does minimal violence to the user's code, reflects the abstract plan, and fixes the greatest number of violations.

4.7.1 Plausibility on the Basis of Side Effects

The first criterion for judging the success of a partial debugging strategy is an analysis of the side effects of the corrections. The debugging strategy with maximal beneficial side effects is preferred. Beneficial side effects occur by eliminating additional model violations, satisfying planning expectations or eliminating violations of rational form.

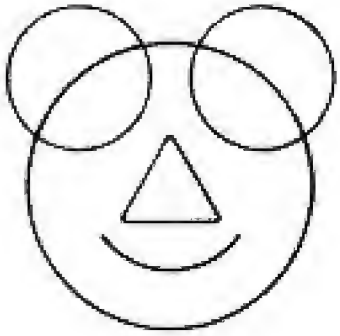
> One might ask why an edit might have any beneficial side effects at all. Isn't it more likely to have bad side effects and cause other violations? The answer is that often several violations are caused by the same error in the code. Then one debugging strategy will stand out from its brethren by fixing this error and thereby simultaneously curing several violations.

On the other hand, sometimes a correction causes additional model violations. In this case, either the new violations can themselves be debugged or the debugging strategy must be abandoned. Assumption and protection commentary are used to help in understanding those bad side effects wherein one edit undoes the effect of some other debugging edit. If the bad side effect cannot be eliminated, then the debugging strategy must be rejected. This is the case with a linear debugging of GOOGLY.EYES (figure 4.5).

The eyes cannot be brought into the head by shrinking the interface without causing them to overlap the nose. Thus this debugging strategy eliminates one violation (OVERLAP EYE HEAD) only to introduce another (OVERLAP EYE NOSE). The system is forced to consider non-linear debugging and fix the parts themselves.

4.7.2 Plausibility on the Basis of Minimal Change

Another plausibility criterion is that of minimal change to the user's code. A debugging strategy that changes an input is preferred to



GOOGLY EYES

FIGURE 4.5

one that adds statements; and a strategy that adds statements is in turn preferred to one that deletes them. The rationale is that a repairman should make minimal changes to a system. The goal is to fix the program in harmony with the user's intent, not to redesign it. This caution is further justified by the fact that the system does not fully know the programmer's intent or plan. Hence, it must be hesitant to make major revisions to his program.

4.7.3 Plausibility on the Basis of Caveat Comments

A third basis for choosing between alternative debugging strategies is advice from the annotator and plan-finder on likely errors. The annotator alerts the debugger to oddities in program structure which may be the underlying cause of some semantic violation (section 2.4). The plan-finder fulfills the same purpose with respect to code that contradicts expectations arising from the type of plan. The mechanism of informing the debugger of the possibly erroneous code is through "caveat" comments. The comments are noticed when the debugger considers the associated code in the course of debugging some model violation. A repair edit is accorded extra plausibility by the

debugger if the correction eliminates the complaint that initiated the caveat.

Caveats generated by the plan-finder are created by noting insertions which are not transparent, interrupted-steps which depend on specific runtime environments and linear plans in which main-steps use the same resource such as an assumption about a particular state variable. In an extended system, caveats would be generated by such oddities as iterative programs which fail to halt and shared free variables. As an example, recall that the arms in NAPOLEON represented a non-transparent insert and that this information advised the debugger to edit the correction into VEE rather than directly into the NAPOLEON super-procedure.

Comments are used--rather than the Annotator or Plan-Finder immediately calling the Debugger to correct the violation--because a violation of rational form is not a guarantee of a bug: the oddity may be harmless or even intended by the programmer. An example in which a sequence of FORWARD instructions arises naturally is the following triangle program:

```
TO TRI
10 FORWARD 50
20 FORWARD 50
30 RIGHT 120
40 FORWARD 100
50 RIGHT 120
60 FORWARD 100
END
```

The first two FORWARD's are surprising. However, if this TRI is being debugged in preparation for being converted to a triangular head with the remainder of the stick-man inserted as statement 15, then the apparent violation of rational form is explained. The utility of comments is that if the code is not suspected of being in error by the

debugger, the comment has no effect. It plays a role only if DEBUG finds a model violation that can possibly be corrected by changing the odd code. Only then does the comment enter into the analysis by supporting such adding plausibility to debugging strategies that eliminate its complaint of non-transparent insert or sequential commands.

4.7.4 Guessing the Culpable Interface

Even with the restriction to linear edits, fixing a predicate relating two main-steps may produce many possible edits. For example, making the head above the legs in NAPOLEON could be done by adding a rotation at any of several places in the program preceding the execution of the TRICORN sub-procedure. Consequently, the system initially considers edits to only two interfaces -- the interface immediately preceding the second main-step (i.e. code for the model part accomplished last) and the initial setup to the program. The immediate interface is preferred on the expectation that preceding interfaces have already been protected in the course of debugging. The global setup is considered because "Unexpected Runtime Environment" is a common cause of errors. The plausibility of these editing points is then analyzed by the criteria described in the preceding sections -- beneficial side effects, minimal change, and caveats as well as the protection criteria described in the preceding section. If they are found implausible, additional interfaces are considered in order, proceeding backwards from the second main-step.

4.8 SUMMARY OF DEBUGGING CONCEPTS

The debugger's knowledge divides into two categories: general debugging technique and specific imperative knowledge of logic and geometry.

Debugging Technique

1. Linear Attack -- First verify main-steps privately. Then analyze relations in terms of interfaces. Only if all else fails, modify main-steps to fix relations.

2. Plausible Search -- Compare alternative debugging strategies using plausiblity criteria of minimal change to the user's code and maximal beneficial side effects.

3. Culpable Interfaces -- Prefer either the initial interface or the interface immediately preceding the bugged module. This is based on the assumption that the temporal attack has already verified intermediate interfaces.

4. Caveats -- Use caveat comments generated by the Plan-Finder and Annotator to suggest the location of the repair.

5. Intrinsic versus Extrinsic Errors -- Classify model violations as intrinsic or extrinsic on the basis of whether the error is internal to the code being examined. Intrinsic errors have limited scope and can be debugged privately.

6. Handling Multiple Bugs -- debug those violations of most-limited scope first: that is, debug properties before relations; then intrinsic predicates before extrinsic ones, and finally in temporal order.

7. Commentary -- Use commentary to express the purpose, assumptions and scope (protection) of a correction and to notice conflicts between different corrections.

Knowledge of Geometry and Logic

1. Imperative Semantics of Predicates - In addition to standard verification code, primitives have semantics that suggest what to do to make the predicate come true. This consists of procedural knowledge which examines code and generates edits to make a particular geometric predicate true.

2. Rigid Body Theorem - This theorem is a precise statement of the effect of the initial environment on a segment of code for Fixed-Instruction Turtle Programs, namely that the code produces a rigid body and that the initial environment affects only the orientation and position.

3. Imperative Knowledge for Logical Predicates - Procedures for making conjunction, disjunction, negation, equality and set membership true with minimal effort.

4.9 Classification of Bugs

The following taxonomy of bugs summarizes the types of errors which the system corrects.

Linear Main-Step Failure:
Manifestation: Failure of main-step to accomplish model part in private, i.e. when run independently.
Fix: (Private Debugging) Repair in private, rejoin and satisfy any initial assumptions.
Ex: (NOT (EQUITRI TRICORN)) in NAPOLEON.

Preparation Error:
Manifestation: Violation of relation between model parts.
Fix: (Linear Debugging) Find culpable interface, make edit suggested by the imperative semantics for the predicate, and protect assumptions and behavour until the point at which the error was manifest.
Ex: See Unexpected Runtime Environment and Local Preparation Errors

Unexpected Runtime Environment: (type of preparation failure)
Manifestation: Violation due to false assumptions of the entry state to program. (Program does succeed in certain environments).
Fix: Add an initial setup which converts the actual entry state to the desired entry state.
Ex: (NOT (BELOW LEGS ARMS)) in NAPOLEON.

Local Preparation Error: (type of preparation error)
Manifestation: Violation intrinsic to the program, and not dependent on the initial environment.
Fix: Modify state appropriate to the imperative semantics for the violated predicate.
Ex: (NOT (VERTICAL TRUNK)) in TREE4.

Non-Linear Main-Step Failure:
Manifestation: Main-step succeeds in private.
Fix: See resource conflicts, insertion errors, and global errors described below.

Unconsidered Second-Order Constraint on Main-step:
(type of non-linear main-step failure)
Manifestation: Violation of a property of model part not detected in private. Manifested by analysis of a relation between the main-step and some other model part.
Fix: Modify main-step in such a way that violation is corrected while the first-order description of properties asserted in the model is still satisfied. Guidance is provided by the imperative semantics for the predicate. Examples of such transformations are dilation and reflection.
Ex: (NOT (INSIDE MOUTH HEAD)) in BIG.MOUTH.

Resource Conflict: (type of non-linear main-step failure)
(Mentioned for completeness: not handled by debugger.)
Manifestation: Violation of property of part described in model which was not exhibited in private.
Fix: Some assumption made when run privately is being violated in public. Such an assumption could be the availability of a given resource, e.g. a free variable.
Ex: Attempt to correct both (VERTICAL BODY) and (HORIZONTAL (SIDE TOP)) in TREE4 by modifying the initial interface statement 5 (section 4.6)

Insertion Error: (type of non-linear main-step failure)
Manifestation: Main-step failure not indicated in private with the additional element that a caveat comment generated by the plan-finder informs the debugger that the code group for the main-step surrounds an insert which is not transparent.
Fix: Make insert state-transparent.
Ex: (NOT (LINE BODY)) in NAPOLEON.

Global Error:
Manifestation: Model part accomplished non-locally fails.
Fix: Find relevant theorem which was the basis of expecting the global plan to succeed. Find assumptions made by theorem which were not justified. Make these assumptions true.
Ex: (NOT (LINE (SIDE 1 TRICORN))) in NAPOLEON.

5. CONCLUSIONS

5.1 TOP-LEVEL DEBUGGING GUIDANCE

The top-level organization of model-driven debugging is to order the model violations and then proceed to fix them in turn. This technique makes the basic assumption that guidance in fixing the program can be obtained by analyzing the specific details wherein the picture failed to satisfy its description. Alternatively, top-level guidance can be obtained through:

1. structure-driven debugging - insight into the form of programs, e.g. such structural considerations as recursive and iterative control patterns and global versus local variable scope.

2. evolution-driven debugging - the evolutionary or editing history of the user's code.

3. process-driven debugging - the abstract form of the process at the time of the error [Sussman 1973].

A more complete debugging system would exhibit all of these forms of direction.

5.2 GENERALIZABILITY OF DEBUGGING TECHNIQUES

The mini-world of programs against which this analysis of debugging is tested is that of fixed-instruction turtle procedures. These are, of course, a particularly simple form of program. Their simplicity allows the imperative semantics for the geometric primitives to utilize the Rigid Body Theorem, justifying the same state change to different interfaces to correct a given bug.

The debugging techniques used to handle even these simple programs are by no means exhaustive. Nevertheless, it is worth noting that many of the techniques utilized by the model-driven debugger are of broad application: an initially linear analysis, the need to order the

attack on multiple bugs, competence to cope with alternative debugging strategies--these are useful regardless of the nature of the top-level direction or the complexity of the program.

The choice of plane geometry as the semantic domain for MYCROFT was not accidental. Geometry allows the use of a Cartesian annotator and a powerful model language for specifying spatial relations. Other domains may not be susceptible to a MYCROFT-like approach because of the lack of powerful ways in which to document the effects of the program and the lack of a good model language. However, it is worth noting two points:

1. spatial models are very important for programming in applications beyond graphics. (This is reflected in the way programmers refer to memory, stacks and data structures in spatial ways.)

and 2. program planning and debugging involve techniques of broad applicability but cannot be entirely done in the absence of domain-dependent knowledge.

5.3 EXTENSIONS

The design of MYCROFT required an investigation of fundamental problem solving issues including description, simplification, linearity, planning, debugging and annotation. MYCROFT, however, is only a first step in understanding these ideas. Further investigation of more complex programs, and of the semantics of different problem domains is necessary. It is also essential to analyze additional planning concepts such as ordering, repetition and recursion as well as the corresponding debugging techniques. Ultimately, such research will surely clarify the learning process in both men and machines by providing an understanding of how they correct their own procedures.

6. BIBLIOGRAPHY

[Abelson 1973]
Abelson, H., Goodman, N. and Rudolph, L.
LOGO manual
LOGO Memo 7 LOGO Project, MIT AI Laboratory (August 1973)

[Floyd 1967]
Floyd, R. W.
"Assigning Meaning to Programs"
Proc. Symp App. Math AMS vol. XIX (1967)

[Fahlman 1973]
Fahlman, Scott
A Planning System For Robot Construction Tasks
AI-TR-283 MIT AI Laboratory (May 1973)

[Goldstein 1974]
Goldstein, I.
Understanding Simple Picture Programs
AI-TR-294 MIT AI Laboratory (March 1974)

[Hewitt 1971]
Hewitt, C.
"Procedural Embedding of Knowledge in PLANNER"
Proc. IJCAI 2 (Sept 1971)

[McDermott 1972]
McCermott, D.V. and G.J. Sussman
The CONNIVER Reference Manual
AI Memo 259 MIT AI Laboratory (July 1973)

[Naur 1967]
Naur, P.
"Proof of Algorithms by General Snapshots"
BIT 6, 1967, 310-316.

[Papert 1971]
Papert, Seymour A.
"Twenty Things to Do with a Computer"
AI Memo 248, MIT AI Laboratory (June 1971)

[Papert 1972]
Papert, Seymour A.
"Teaching Children Thinking"
Programmed Learning and Educational Technology, Vol.9, No.5 (Sept 1972)

[Sussman 1970]
Sussman, G.J., T. Winograd, and E. Charniak
Micro-Planner Reference Manual
AI Memo 203, MIT AI Laboratory (December 1971)

[Sussman 1973]
Sussman, G.J.
A Computational Model of Skill Acquisition
AI-TR-297 MIT AI Laboratory (Sept 1970)

[Winston 1970]
Winston, P.H.
Learning Structural Descriptions from Examples
MAC-TR-76 MIT AI Laboratory (Sept 1970)